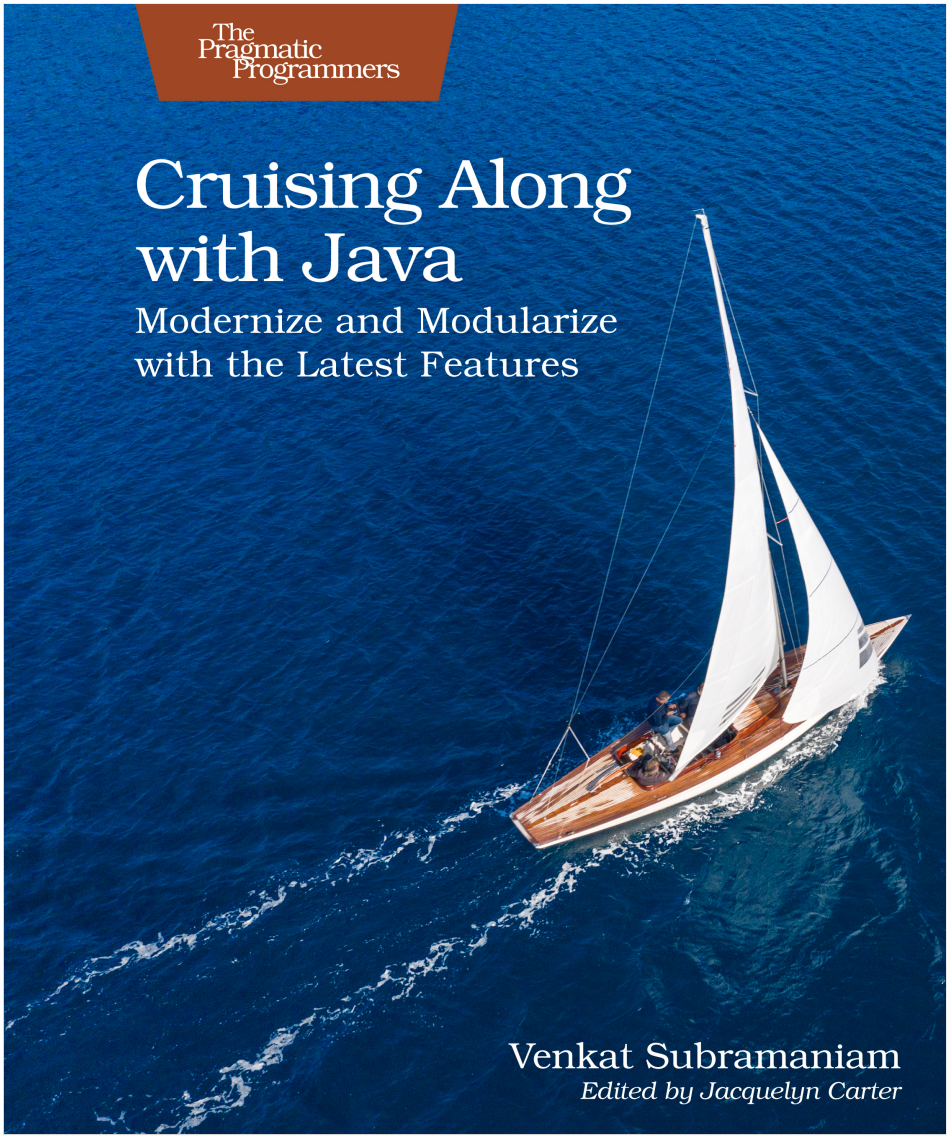

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Representing data in code is a common task that has been a real chore in Java. Whether you were creating an XML document in code, generating a JSON response to a web request, or creating a nicely formatted customized message as an automated response from your support system, the code often was verbose, smelly, hard to read, and difficult to maintain. The coding experience was rather unpleasant largely due to the inability to write multiple lines of strings with ease and the endless escape sequences that had to be placed in the strings. These were tasks any Java programmer dreaded…until recently.

Some shells and programming languages offer *heredocs* as a feature to deal with escapes but often have rough edges when dealing with indentations and text termination. Programmers using heredocs often find it frustrating and waste time due to idiosyncrasies of implementations. The designers behind the evolution of Java took advantage of learning from the earlier solutions in other platforms and languages. The result is a pleasant experience for the Java programmers.

The text blocks feature was introduced in Java 13 and has evolved over a few versions of the language. With text blocks, we can write multiple lines of text with ease and don't have to waste our time and effort with noisy escape sequences. The text flows naturally, and the compiler is smart enough to discern between the indentations in code and those in the text. The compiler is also capable of recognizing and omitting unintended trailing spaces in text and thus removes the need to strip them out from text placed in code. Overall, the smartness of the implementation leads to better developer productivity.

In this chapter we'll look at the problems that text blocks solve and at how to make use of this feature to embed raw text, XML, and JSON data in code. You'll learn about the behavior of text blocks and the new escape sequences. Along the way, we'll also take a peek at the implementation of text blocks at the bytecode level.

Let's explore text blocks by starting with an example that suffers from verbosity, and then we'll refactor the code to make it expressive and elegant.

From Noisy to Nice

Suppose you're working on an application for an online retailer and the task on hand requires creating a message that will be emailed to users, asking for their feedback by filling out a survey.

The message is expected to be of the following format right now but may change in the future to add user and purchase-specific details:

```
Thank you for your purchase. We hope you had a pleasant experience.

We request that you take a few minutes to provide your feedback.

Please fill out the survey at https://survey.example.com

If you have any questions or comments, please click on the "Support" link
at https://www.example.com.
```

In the older versions of Java, you may have to write code like the following to create the message:

textblocks/vsca/CreateMessage.java

```
public static String createMessage() {
  String message = "Thank you for your purchase.";
  message += " We hope you had a pleasant experience.\n\n";
  message += "We request that you take a few minutes ";
  message += "to provide your feedback.\n\n";
  message += "Please fill out the survey at https://survey.example.com\n\n";
  message += "If you have any questions or comments, ";
  message += "please click on the \"Support\" link\n";
  message += "at https://www.example.com.\n";

  return message;
}
```

The code uses += to append the text to the String instance. We could have replaced message += with + to reduce some noise. The code uses combinations of \n to provide line breaks and uses escape to include double quotes in the text. Also, each line has to end with a semicolon, adding to the noise.

That's one verbose code…*shudder*…one you'd hide for the sake of humanity, definitely not one you would show to children. I bet that += isn't a feature you'd put on your resume either. We need better. Thankfully, Java has us covered, starting from version 13.

You can refactor the noisy code with text blocks and make it nice and concise, like so:

textblocks/vsca/CreateMessageConcise.java

```
public static String createMessage() {
  var message = """
    Thank you for your purchase. We hope you had a pleasant experience.

    We request that you take a few minutes to provide your feedback.

    Please fill out the survey at https://survey.example.com

    If you have any questions or comments, please click on the "Support" link
    at https://www.example.com.
    """;

  return message;
}
```

The refactored version produces exactly the same output as the noisy version, but the code is easier to read and doesn't use +=. Also, there are no escapes for double quotes and no smelly line breaks.

To create this code, you may literally copy the text from a requirements document, paste it into code, and add the necessary syntax before and after to define a text block. It's a huge win to go from the requirements to code with such little effort.

A text block starts with three double quotes """ followed by a line terminator—they're truly intended for multiline strings. A text block ends also with three double quotes """, but that may appear on the same line as the ending text or on a new line—see Smart Indentations, on page ?.

Before we dig further into text blocks, we should quickly take a look at how they're implemented at the bytecode level. Knowing this will help us to answer questions that developers often ask about the effect of text blocks on performance, serialization, and interoperability with other languages.

Text blocks are purely a Java compiler feature and don't have any special representation in the bytecode. Once the compiler processes the indentation and escape characters, it creates a regular String. We can confirm this by running the javap tool on the bytecode. Let's take a look at the bytecode generated for the previous createMessage() method that uses a text block:

textblocks/shoutput/runCreateMessageConcise.sh.output

```
...
  public static java.lang.String createMessage();
    Code:
       0: ldc           #7                  // String Thank
you for your purchase. We hope you had a pleasant
experience.\n\nWe request that you take a few minutes to
provide your feedback.\n\nPlease fill out the survey
at https://survey.example.com\n\nIf you have any
questions or comments, please click on the \"Support\"
link\nat https://www.example.com.\n
...
```

If we take a quick look at the details produced by the javap tool, we see that the bytecode has instructions to load up a constant (ldc) value of a String. The String contains the data created within the text block, with necessary escapes added in for proper formatting.

There is no runtime impact to process text blocks; the compiler does the heavy lifting. There are no serializability issues since the representation is the good old String and it's intended to provide the same performance benefits we've

enjoyed all along. There is no interoperability issue either since at runtime there is no concept of text blocks—it's all merely Strings.

In addition to removing the need to concatenate texts using + or +=, Java removes the need to use most escape characters when building a string. Let's take a look at that capability next.

Embedding Strings

To embed a double quote within a string we have to use escape characters. This will result in bloated code that's hard to maintain, especially when working with code to generate XML or JSON documents. Text blocks remove all that noise by letting us place single and double quotes freely within a string. Let's look at the benefit of this feature with an example.

Suppose we're asked to create code to generate the following text:

```
The 'National Weather Service' has issued a "severe" thunderstorm warning
for tomorrow. Please """stock up""" on the essentials you'll need during
the adverse weather.

\Approved for general distribution\
```

To create this text using the common string, we may litter the code with escape sequences, like so:

textblocks/vsca/Escapes.java

```
String message = "The \'National Weather Service\' has issued a " +
  "\"severe\" thunderstorm warning\nfor tomorrow. " +
  "Please \"\"\"stock up\"\"\" on the essentials you'll need " +
  "during\nthe adverse weather.\n\n\\Approved for general distribution\\";
```

Good code should be inviting to the reader's eyes. The noise of escapes will likely dissuade even the most excited programmer eager to maintain the code. It takes a lot of effort to change the code in these situations, and you can forget about copying and pasting text directly from the requirements document to code.

Thanks to text blocks, we can remove most of the noise from the previous code.

```
var message = """
  The 'National Weather Service' has issued a "severe" thunderstorm warning
  for tomorrow. Please \"""stock up\""" on the essentials you'll need during
  the adverse weather.

  \\Approved for general distribution\\""";
```

Since three quotes are used as a delimiter for text blocks, in the rare occasion when three double quotes appear continuously in the text, we'll have to escape,

but with a single backslash. Also, since backslash is used as an escape character, we'll have to escape that with another backslash if it appears in the text.

As you can see, the multiline text block can handle raw strings with less clutter, is effortless to read, easy to change, and is convenient to copy and paste from other sources into code.

Multiline strings aren't unique to Java; they exist in other languages. But one of Java's innovations is how it handles the indentation of the text. Let's dive into that next.